CSE 586 – Distributed Systems
GitHub Notification System
Project 1: Phase 2
Documentation

Team number: 78
Siddharth Sankaran (50421657)
Shriram Ravi (50419944)

23 October 2021

1. Overview

This purpose of this document is to provide a detailed understanding of the publish-subscribe distributed system that has been implemented as part of Project-1 of the Distributed Systems course. This document will talk about the design, data flow and brief overview of the steps to implement the code.

The system implemented here is a GitHub Notification System that allows a user to receive notifications from their preferred repositories. The user subscribes to different repositories available in GitHub by specifying the owner, repository, etc. in the front-end web application. The user then receives notifications of the latest commits in that repository.

The system also allows the publisher to send advertisements to the users through a simple user interface. Hence every time a user logs in, they can see the advertisements published by the publisher. The publisher can de-advertise to stop the publishing of the advertisement.

Users can freely subscribe and unsubscribe from repositories from time to time making the system dynamic and intuitive.

2. Design

The system is implemented with 4 components:

2.1. Publisher

The publishers in this implementation are the different repositories in GitHub that the user has subscribed to. Every commit to the subscribed repository will be published in the database and

will trigger a notification to the user. The publisher end is implemented through a Python Flask application. The GitHub public API is called with specific arguments for owner, repo, time to receive the latest commit messages from the repo since the specified time. The publisher pushes this information (publish() function) to the middleware (broker) which then inserts it into the database. The publisher is dockerized and run as a separate container that performs periodic API calls and pushes information to the middleware.

2.2. Subscriber

The subscribers in this implementation will log into the web application with a username and can subscribe to any public repository available in GitHub. This is enabled through a simple front end Flask application that allows the user to pick the repository they want and add a subscription. This pushes the subscriber information to the middleware (subscribe() function) which inserts it into the dataset.

Based on the list of subscriptions of a user, the front end receives appropriate notifications from the middle ware periodically when new information is published. The front end has a notifications tab that allows the user to see the latest updates in each repository.

The front end also has tabs to add new subscriptions and unsubscribe from existing topics.

2.3. Middleware

In this implementation, the middleware is the intermediary that is responsible for enabling indirect communication between the publisher and the subscriber. The middleware is also the only entity in the system that can interact with the database. Accordingly, the middleware is equipped to receive data from the front end for the following requests: new subscribe, new unsubscribe, new login, new advertise, new de-advertise. Similarly, it also handles push requests from the publisher end containing the data of latest commits. The middleware is responsible for updating the database accordingly.

The middleware is also responsible for pushing notifications (notify()) to each user when the user is online. This is done with a simple online flag that is turned on when the user logs in. The system then performs matching to match the user's subscription list and the latest update seen by the user with the list of messages from the database to pick out the right set of unique

notifications to each user. This notification is pushed to the frontend notifications tab that can be viewed on refreshing the webpage.

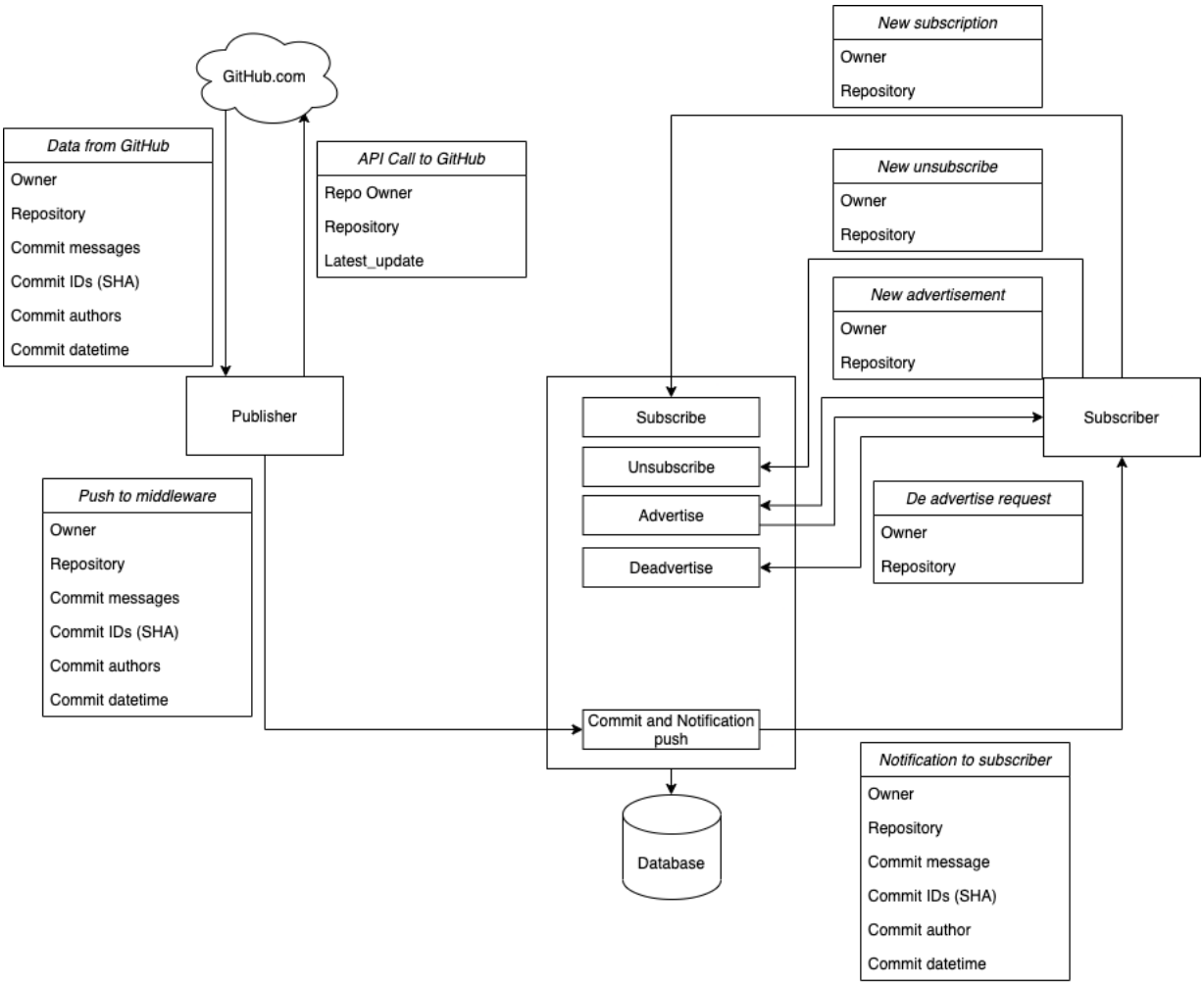
The middleware is containerized separately and is deployed along with the database using the docker-compose command.

2.4. Database

The database used in this system is MongoDB. Each database is organized into collections and documents. The database created for this example is subscribers_db which is composed of 3 collections

a. subscribers_db – Contains list of subscribers with their subscription list. Each subscription list contains the list of repos and the latest update seen by the user
b. topics_db – Unique list of topics that all users are subscribed to. Also contains the latest update of each topic that is present in the database
c. commit_messages_db – Set of commit messages for each user subscribed topic

3. Dataflow Diagram



The above diagram depicts all the data transactions in this project.

On the left-hand side of the diagram is the publisher end. Here the publisher flask application polls the GitHub public API to obtain the recent messages from the all the subscribed topics. This returns the message and message related information like id, timestamp, author, etc. to the publisher. The publisher passes this on to the middleware through a POST http request to the middle docker container, specifically to the commit_notif_push() method

The middleware receives the new messages and inserts them into the MongoDB using the insert_one() method from the PyMongo module. Then this method also begins performing the matching operation every time a new data is pushed to the data base. The matching operation interacts with the database to find the list of subscribers online, fetch their respective

subscriptions and filter the latest messages. The notification for each user is filtered based on the last message seen by the user for each of his subscription. This notification is then pushed to the frontend with a POST request to the notifications page. Once the notification is sent each user's latest update timestamp is updated with the time of the latest commit. Each topic's latest commit time is also updated in the database.

In the right-hand side the subscriber inputs data in the form of new subscription, specifying owner name and repository. This is handled by the subscribe function which inserts this information into the database, into subscribers_db. A similar workaround for the unsubscribe functionality is also implemented.

For creating an advertisement, the user can visit the /advertise route in the web application to create a new advertisement. This info is sent to the middleware which modifies the advertise flag in the database to 1, enabling a rolling advert in each webpage. De-advertise is performed by moving to the de-advertise tab in this page. This will send a POST request to the middleware, which interacts with the database to change the advertise flag to 0, thus removing the rolling advertisement from each page.

4. Steps to run the code

Please find below the steps to be followed to obtain a successful implementation of Phase 2 of the project

a. In Terminal, navigate to frontend folder and run to create the frontend docker image
   - docker build -t frontend-image .
b. Use the image to create the front-end container
   - docker run --name frontend-container -p 5000:5000 frontend-image
   - You can create multiple instances of front-end containers (Subscribers). These needs to be exposed to different local ports.
c. This will run the frontend container on port 5000 in localhost
d. In another terminal, navigate to the api_call folder and run the following command

- docker build -t apicall-image .
- docker run --name apicall-container -p 5002:5002 apicall-image
- This will create a container designed for fetching publishers' data from the GitHub

e. In another terminal, navigate to the backend folder and run the following command
   - docker-compose up
   - This will create separate containers for the middleware (flask server) and the backend (MongoDB) and initiate them. Both these containers are placed in a docker network
f. In another terminal, run the following command to add the front-end container and the api_call container to the above docker network so that all three containers are in the same network
   - docker network connect backend_default frontend-container
   - Do the same for all the subscriber containers to connect them to the docker network
   - docker network connect backend_default apicall-container
g. To verify if all three containers are in the same network run,
   - docker network inspect backend_default
h. Open a browser and go to
   - localhost:5002
   - This will start the data fetcher container and update the appropriate GitHub events to the DB
i. Open a browser and go to
   - localhost:5000
   - Provide appropriate inputs. These inputs will be added to the MongoDB once the submit button is clicked
   - This is extensible to all the containers created with the frontend-image